

Grundlagen der Künstlichen Intelligenz

10. Klassische Suche: Breitensuche

Malte Helmert

Universität Basel

13. März 2015

Klassische Suche: Überblick

Kapitelüberblick klassische Suche:

- 5.–7. Grundlagen
- 8.–12. Basialgorithmen
 - 8. Datenstrukturen für Suchalgorithmen
 - 9. Baumsuche und Graphensuche
 - 10. Breitensuche
 - 11. uniforme Kostensuche
 - 12. Tiefensuche und iterative Tiefensuche
- 13.–20. heuristische Algorithmen

Blinde Suche

Blinde Suche

In Kapitel 10–12 betrachten wir **blinde** Suchalgorithmen:

blinde Suchalgorithmen

Blinde Suchalgorithmen verwenden **keine** Informationen über Zustandsräume ausser dem Black-Box-Interface. Sie werden auch **uninformierte** Suchalgorithmen genannt.

vergleiche: **heuristische** Suchalgorithmen (ab Kapitel 13)

Blinde Suchalgorithmen: Beispiele

Beispiele für blinde Suchalgorithmen:

- Breitensuche
- uniforme Kostensuche
- Tiefensuche
- tiefenbeschränkte Suche
- iterative Tiefensuche

Blinde Suchalgorithmen: Beispiele

Beispiele für blinde Suchalgorithmen:

- **Breitensuche** ($\rightsquigarrow$ dieses Kapitel)
- uniforme Kostensuche
- Tiefensuche
- tiefenbeschränkte Suche
- iterative Tiefensuche

Blinde Suchalgorithmen: Beispiele

Beispiele für blinde Suchalgorithmen:

- **Breitensuche** (↪ dieses Kapitel)
- uniforme Kostensuche (↪ Kapitel 11)
- Tiefensuche (↪ Kapitel 12)
- tiefenbeschränkte Suche (↪ Kapitel 12)
- iterative Tiefensuche (↪ Kapitel 12)

Breitensuche: Einführung

Breitensuche

Breitensuche expandiert Knoten **in Erzeugungsreihenfolge** (FIFO).

↪ z. B. Open-Liste als **verkettete Liste** oder **Deque**

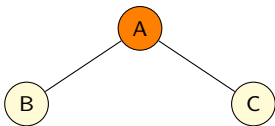


open: **A**

Breitensuche

Breitensuche expandiert Knoten **in Erzeugungsreihenfolge** (FIFO).

↪ z. B. Open-Liste als **verkettete Liste** oder **Deque**

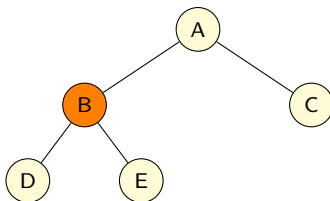


open: **B**, C

Breitensuche

Breitensuche expandiert Knoten **in Erzeugungsreihenfolge** (FIFO).

↪ z. B. Open-Liste als **verkettete Liste** oder **Deque**

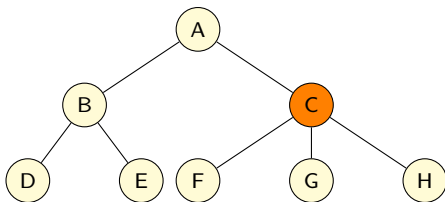


open: **C**, D, E

Breitensuche

Breitensuche expandiert Knoten in **Erzeugungsreihenfolge** (FIFO).

↪ z. B. Open-Liste als **verkettete Liste** oder **Deque**

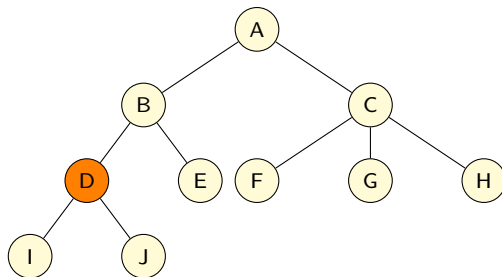


open: **D**, E, F, G, H

Breitensuche

Breitensuche expandiert Knoten **in Erzeugungsreihenfolge** (FIFO).

↪ z. B. Open-Liste als **verkettete Liste** oder **Deque**

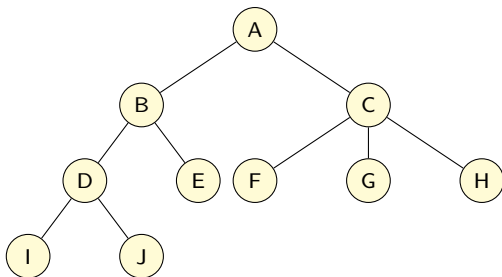


open: **E**, F, G, H, I, J

Breitensuche

Breitensuche expandiert Knoten **in Erzeugungsreihenfolge** (FIFO).

↪ z. B. Open-Liste als **verkettete Liste** oder **Deque**



- durchsucht Zustandsraum **ebenenweise**
- findet immer **flachsten** Zielzustand zuerst

Breitensuche: Baumsuche oder Graphensuche?

Breitensuche kann

- **ohne Duplikateliminierung** (als Baumsuche)
 $\rightsquigarrow$ **BFS-Tree**
- **oder mit Duplikateliminierung** (als Graphensuche)
 $\rightsquigarrow$ **BFS-Graph**

durchgeführt werden (BFS = **breadth-first search**).

$\rightsquigarrow$ Wir betrachten beide Varianten.

BFS-Tree

Erinnerung: generischer Baumsuchalgorithmus

Erinnerung aus Kapitel 9:

Generische Baumsuche

```
open := new OpenList
open.insert(make_root_node())
while not open.is_empty():
    n := open.pop()
    if is_goal(n.state):
        return extract_path(n)
    for each  $\langle a, s' \rangle \in \text{succ}(n.state)$ :
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable
```

BFS-Tree (1. Versuch)

Breitensuche ohne Duplikateliminierung (1. Versuch):

BFS-Tree (1. Versuch)

```
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    if is_goal(n.state):
        return extract_path(n)
    for each  $\langle a, s' \rangle \in \text{succ}(n.\text{state})$ :
        n' := make_node(n, a, s')
        open.push_back(n')
return unsolvable
```

BFS-Tree (1. Versuch): Diskussion

Das ist schon fast ein brauchbarer Algorithmus,
aber er verschwendet etwas Zeit:

- In einer Breitensuche ist der erste erzeugte Zielknoten auch immer der erste expandierte Zielknoten. (Warum?)
- Daher ist es effizienter, den Zieltest bereits durchzuführen, wenn ein Knoten erzeugt wird (nicht erst, wenn er expandiert wird).

↪ Wieviel Zeit spart das?

BFS-Tree (2. Versuch)

Breitensuche ohne Duplikateliminierung (2. Versuch):

BFS-Tree (2. Versuch)

```
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    if is_goal(n.state):
        return extract_path(n)
    for each  $\langle a, s' \rangle \in \text{succ}(n.\text{state})$ :
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Tree (2. Versuch): Diskussion

Wo ist der Bug?

BFS-Tree (endgültige Version)

Breitensuche ohne Duplikateliminierung (endgültige Version):

BFS-Tree

```
if is_goal(init()):  
    return  $\langle \rangle$   
open := new Deque  
open.push_back(make_root_node())  
while not open.is_empty():  
    n := open.pop_front()  
    for each  $\langle a, s' \rangle \in \text{succ}(n.\text{state})$ :  
        n' := make_node(n, a, s')  
        if is_goal(s'):  
            return extract_path(n')  
        open.push_back(n')  
return unsolvable
```

BFS-Tree (endgültige Version)

Breitensuche ohne Duplikateliminierung (endgültige Version):

BFS-Tree

```
if is_goal(init()):  
    return ⟨⟩  
open := new Deque  
open.push_back(make_root_node())  
while not open.is_empty():  
    n := open.pop_front()  
    for each ⟨a, s'⟩ ∈ succ(n.state):  
        n' := make_node(n, a, s')  
        if is_goal(s'):  
            return extract_path(n')  
        open.push_back(n')  
return unsolvable
```

BFS-Graph

Erinnerung: generischer Graphensuchalgorithmus

Erinnerung aus Kapitel 9:

Generische Graphensuche

```
open := new OpenList
open.insert(make_root_node())
closed := new ClosedList
while not open.is_empty():
    n := open.pop()
    if closed.lookup(n.state) = none:
        closed.insert(n)
        if is_goal(n.state):
            return extract_path(n)
        for each  $\langle a, s' \rangle \in \text{succ}(n.state)$ :
            n' := make_node(n, a, s')
            open.insert(n')
return unsolvable
```

Anpassung der generischen Graphensuche für Breitensuche

Anpassung der generischen Graphensuche für Breitensuche:

- analoge Anpassungen zu BFS-Tree
(Deque als Open-Liste, früher Zieltest)
- da Closed-Liste hier nur zur Duplikaterkennung dient, nicht zur Verwaltung von Knoteninformationen, reicht eine Mengen-Datenstruktur aus
- aus denselben Gründen, warum frühe Zieltests sinnvoll sind, sollten wir Duplikattests gegen die Closed-Liste und Updates der Closed-Liste so früh wie möglich vornehmen

BFS-Graph (Breitensuche mit Duplikateliminierung)

BFS-Graph

```
if is_goal(init()):  
    return ⟨⟩  
open := new Deque  
open.push_back(make_root_node())  
closed := new HashSet  
closed.insert(init())  
while not open.is_empty():  
    n := open.pop_front()  
    for each ⟨a, s'⟩ ∈ succ(n.state):  
        n' := make_node(n, a, s')  
        if is_goal(s'):  
            return extract_path(n')  
        if s' ∉ closed:  
            closed.insert(s')  
            open.push_back(n')  
return unsolvable
```

Eigenschaften der Breitensuche

Eigenschaften der Breitensuche

Eigenschaften der Breitensuche:

- BFS-Tree ist **semi-vollständig**,
aber nicht **vollständig** (Warum?)
- BFS-Graph ist **vollständig** (Warum?)
- BFS (beide Varianten) ist **optimal**,
wenn alle Aktionen dieselben Kosten haben (Warum?),
aber nicht im allgemeinen Fall (Warum nicht?)
- Aufwand: **folgende Folien**

Breitensuche: Aufwand

Das folgende Ergebnis gilt für beide BFS-Varianten:

Satz (Zeitaufwand der Breitensuche)

Sei b der maximale Verzweigungsgrad und d die minimale Lösungslänge des gegebenen Zustandsraums. Sei $b \geq 2$.

*Dann beträgt der **Zeitaufwand** der Breitensuche*

$$1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$$

Erinnerung: wir messen Zeitaufwand in erzeugten Knoten

Es folgt, dass der **Speicheraufwand** beider BFS-Varianten ebenfalls $O(b^d)$ ist (falls $b \geq 2$). (**Warum?**)

Breitensuche: Beispiel für den Aufwand

Beispiel: $b = 10$; 100'000 Knoten/Sekunde; 32 Bytes/Knoten

d	Knoten	Zeit	Speicher
3	1'111	0.01 s	35 KiB
5	111'111	1 s	3.4 MiB
7	10^7	2 min	339 MiB
9	10^9	3 h	33 GiB
11	10^{11}	13 Tage	3.2 TiB
13	10^{13}	3.5 Jahre	323 TiB
15	10^{15}	350 Jahre	32 PiB

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

Vorteile von BFS-Graph:

- vollständig
- viel (!) effizienter, wenn es viele Duplikate gibt

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

Vorteile von BFS-Graph:

- vollständig
- viel (!) effizienter, wenn es viele Duplikate gibt

Vorteile von BFS-Tree:

- einfacher
- weniger Overhead (Zeit/Platz), wenn wenige/keine Duplikate

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

Vorteile von BFS-Graph:

- vollständig
- viel (!) effizienter, wenn es viele Duplikate gibt

Vorteile von BFS-Tree:

- einfacher
- weniger Overhead (Zeit/Platz), wenn wenige/keine Duplikate

Schlussfolgerung

BFS-Graph ist normalerweise zu bevorzugen, es sei denn wir wissen, dass es im gegebenen Zustandsraum vernachlässigbar wenige Duplikate gibt.

Zusammenfassung

Zusammenfassung

- **blinde Suchverfahren:** verwenden keine Informationen ausser Black-Box-Interface des Zustandsraums
- **Breitensuche:** expandiere Knoten in Erzeugungsreihenfolge
 - durchsucht Zustandsraum **ebenenweise**
 - als Baumsuche oder als Graphensuche möglich
 - Aufwand $O(b^d)$ bei Verzweigungsgrad b , minimale Lösungslänge d (falls $b \geq 2$)
 - **vollständig** als Graphensuche; **semi-vollständig** als Baumsuche
 - **optimal** bei **einheitlichen Aktionskosten**